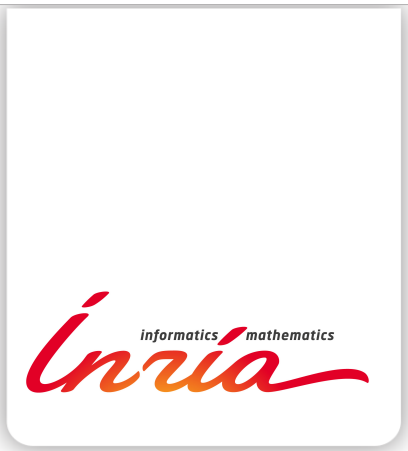
COMET: A High-Performance Model for Fine-Grain Composition

Julien Bigot , Thierry Gautier, Christian Pérez, Jérôme Richard

RESEARCH REPORT
N° 9086
May 2017
Project-Teams Avalon

ISSN 0249-6399 ISRN INRIA/RR--9086--FR+ENG



COMET: A High-Performance Model for Fine-Grain Composition

Julien Bigot* †, Thierry Gautier†, Christian Pérez†, Jérôme Richard†

Project-Teams Avalon

Research Report n° 9086 — May 2017 — 21 pages

Abstract: This paper deals with the efficient combination of software components and task-based models for HPC. Task-based models are known to greatly enhance performance and performance portability while component models ease the separation of concerns and thus improves modularity and adaptability. The paper describe the COMET programming model, a component model for HPC extended with task concepts. We demonstrate its prototype implementation built on top of the task model of OpenMP and the low level component model L2C. We evaluate the approach on five synthetic use-cases representative of common patterns from HPC applications. Experimental results show that the approach is very efficient on the use-cases. On one hand, independent software codes can be easily composed. On the other hand, fine-grained composition supports very good performance. It sometimes even outperforms classical hand-written OpenMP implementations thank to better task interleaving.

Key-words: HPC, software component model, task-based model, task scheduling, multi-cores, shared-memory

* Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay
† Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude-Bernard Lyon 1, LIP

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

COMET: Un modèle de haute-performance pour la composition à grain fin

Résumé : Ce rapport traite de la combinaison de modèles à composants et de modèles d'ordonnancement de tâches pour le calcul haute-performance (HPC). Les modèles d'ordonnancement de tâches sont connu pour améliorer les performances et la portabilité des performance des codes HPC tandis que les modèles à composants facilite la séparation des préoccupation et donc améliorent la modularité et l'adaptation des codes. Le rapport décrit le modèle de programmation COMET: un modèle à composant HPC étendu avec des concepts de tâches. Nous démontrons sa mise en œuvre utilisant le modèle de tâche OpenMP ainsi que le modèle à composant de bas niveau L2C. Nous évaluons l'approche sur cinq cas d'utilisation synthétiques représentatif des patrons de codes issues des applications HPC. Les résultats expérimentaux montrent que l'approche se révèle être très efficace sur les cas d'utilisation présentés. D'une part, la composition de des codes indépendants est facilement réalisable. D'autre part, la composition à grain fin permet d'obtenir de très bonne performances. Les performance obtenue avec cette approche sont même parfois meilleures que celle obtenue avec un code OpenMP écrit à la main grâce à une exécution efficace entrelaçant l'exécution des tâches.

Mots-clés : calcul haute-performance, modèle à composants, modèle à base de tâches, ordonnancement de tâches, multi-coeurs, mémoire partagée

1 Introduction

High-performance architectures are very difficult to program efficiently. One major source of complexity comes from the diversity and constant evolution of the architecture of computing nodes. The number of cores per node keeps increasing with deep and complex cache hierarchies and potentially non-uniform main memory accesses, etc.

Programming models evolve to handle this complexity. Many new shared-memory programming models emerge to complement distributed-memory models in what is often referred to as MPI+X. In this context, task-based programming offers a good approach to handle node-level complexity while supporting good performance.

Nonetheless, while many works focus on performance, they often overlook software engineering. Therefore, the maintainability, evolution, or re-use in HPC codes is often low. Huge development costs are generated because of duplication of efforts, multiplication of concerns in the same code, etc.

Hence, HPC code developers are faced with a fundamental dilemma between maintainability and performance. They have to make a difficult choice regarding the best trade-off.

A solution on the software engineering side of the dilemma is brought by component-based software engineering (CBSE) [19, 25]. CBSE proposes to build applications by assembling independent software building blocks (components) with well-defined interfaces. This enables easy reuse of (potentially third-party) components and architectural-level modifications of applications through their assembly.

A previous work [5] has demonstrated the feasibility of combining concepts of task-based models for high performance and component models for software engineering. We have named this the COMET approach and have validated it on a hand written use case extracted from the GYSELA application [16, 17].

This paper extends previous work by presenting the programming model of COMET, an implementation based on OpenMP, as well as experimental evaluations. COMET can be seen as an extension of the HPC-oriented low-level component model L^2C with a dataflow model as a new way of composing components. In the dataflow, code execution is handled by *metatasks* whose parallelism is deduced from partitioned data that can be inputs or outputs of a metatask. The model can take advantage of data repartitioning between metatasks for fine-grain metatask composition.

COMET emphasizes code-reuse and supports separation of concerns while being able to achieve high-performance. The model is restricted to shared memory but can be combined with MPI for inter-node communications thanks to the MPI connectors of L^2C [8].

This paper studies the performance of the COMET compiler that generates OpenMP code and its runtime. Experiments have been done on five synthetic benchmarks to evaluate the overheads and advantages of fine-grain composition. Those benchmarks are inspired from common patterns of HPC applications.

The remainder of this paper is organized as follows. Section 2 deals with related work. Section 3 present the COMET model through some examples while the prototype implementation is described in Section 4. Section 5 evaluates the approach both in terms of software-engineering capabilities and performance on five synthetic benchmarks. Section 6 concludes the paper and gives some perspectives.

2 Related-work

As the base COMET model is a component model, let us first analyze work related to component models. The common component architecture (CCA) [2] and low-level components (L^2C) [8] are

two HPC oriented component models that target to improve maintainability and adaptability of HPC applications with low overhead. In particular they both provide support for MPI. In these two models, components expose services (*i.e.,* a set of functions) using *provide ports* and require services provided by other components through *use ports*. An assembly is a set of component instances whose use ports are connected to provide ports, enabling the first one to calls functions of the second one. Such composition aims at significantly easing code reuse as well as code coupling.

Some CCA extensions [2], as well GridCCM [23], do provide parallel composition but mainly targeting inter-process method invocations based on function argument partitioning. They do not support dataflow like composition. These models do not offer any particular support for expressing task dependencies. They both only rely on sequential method invocation. Indeed, when an algorithm is split into multiple independent components, these models do not provide any simple way to efficiently interleave the parallel execution of the different parts. In fact, this burden is left to application developers

Interleaving of algorithm execution without specification of the exact execution order by developers is traditionally supported by the dataflow paradigm. A variant of this paradigm suited to HPC requirements is offered by task-based models. Task-based runtime systems such as Parsec [27], Legion [6], OmpSs [11], XKaapi [15], StarPU [4] and Peppher [7] enable to efficiently execute computational parts of HPC applications. This is achieved by scheduling computation units called tasks over the available resources (CPU or GPGPU for example). The dataflow is expressed by declaring for each task its input and output data dependencies. In most models, these data dependencies (or flow dependencies) form a directed acyclic graph (DAG). In such a DAG, nodes are tasks and edges represent (data) dependencies between tasks. A DAG is then scheduled by a runtime, targeting an efficient execution over a wide range of hardware. However, most tools only focus on execution-time matters. The software engineering aspects such as code composition or maintainability is quite low and depends on the model used to describe the task-graph.

Approaches such as Regent [24] or Swift [13] offer models to describe applications task-graphs using a dedicated language. Regent [24] and Swift [13] are programming languages for implicit dataflow parallelism. A high-level task-based imperative language offers developers a way to implicitly (*i.e.,* in a transparent way) build their task graph using constructs such as loops, conditionals, etc. Actually, function calls or specific code constructs are transparently replaced during execution by submitting tasks to a task-based runtime. While proposing high-level abstraction to end-users, low-level data-based composition can naturally be expressed using Regent thanks to two given abstractions: tasks, and logical regions (*i.e.,* collections of structured objects that can recursively be partitioned). However, high-level coupling of independent code is not handled in Regent. Moreover, the use of an imperative language to implicitly create the task DAG does not improve application structure understanding, separation of concerns into independent software parts, or ease of replacement and maintainability like component models do through the concept of assembly.

OpenMP [22] is a well-known API that supports shared memory multiprocessing programming. It provides means to easily incorporate parallelism into sequential applications, at a relatively high-level through the use of code annotations. It supports both task and data parallelism. While OpenMP helps to easily parallelize HPC applications, it does not change the underlying programming paradigm of the annotated language (*i.e.* imperative programming). Supporting such an approach for component models, where parallelization aspects could be specified by a dedicated language in the assembly, would be a very interesting feature. However, the authors are not aware of any such work. OpenMP can be used in C, C++ or FORTRAN component implementations but is not available at the assembly level. The approach proposed in this paper

can be seen as a step in this direction.

Data-driven workflows such as Gwendia [20] and dataflow-based models such as FlowVR [3], FastFlow [1], or the model proposed by Lau and al. [18], emphasize easing code composition and also provide a higher abstraction than task-based runtime systems. They enable composing algorithms through data-based composition where components of the models expose data ports which, once connected, enable components to produce data consumed by other ones. Such composition is a convenient way to describe many HPC applications. However, these models are mainly designed for heavy-grained task-based composition. As a result, coupling HPC codes with fine-grain task parallelism using these models would be detrimental to performance and maintainability, as it would require codes to be split into many independent parts.

Spatio-Temporal Component Model [9] (STCM) unifies features from both component models and data-driven workflow models. The model provides composition units called component-tasks (merging tasks and components). Such units can be composed into assemblies through both use-provide and data connections. However, the model does no target fine-grain intra-node parallelism for HPC applications as it does not support data partitioning for example. Components and tasks are merged together, which would result in some overhead if used for fine-grained task decomposition of HPC applications, due to component instantiation and connection overheads.

Thus, end-users have to face a trade-off between maintainability and performance on modern HPC architectures. None of the proposed models are satisfactory as they do not enable the writing of fully independent modules efficiently coupled at runtime.

The COMET approach has been introduced in [5]. The paper shows the feasibility of efficiently combining both a software component model and a task-based model on a use case extracted from the GYSELA application. However, the model is very briefly stacked as the paper focuses on runtime issues to evaluate the feasibility. As a consequence, the programming model is limited to the features needed to the use case. In particular, data repartitioning was not supported.

The next section describes the COMET programming model with its support for data partitioning and repartitioning as well as its support for task data alignment.

3 The COMET Model

This section describes COMET, a HPC component model with dataflow-like composition that emphasizes simple and efficient parallel code composition. The model targets shared-memory and relies on MPI for inter-node composition. It is based on the minimalist HPC-oriented component model L^2C [8].

COMET extends L^2C with support for dataflow-based interactions between components and metatasks (a special kind of component.) Such a fine-grain dataflow composition supports the creation of task graphs. In the remaining of this section, the elements of the COMET model are defined through some examples using a pseudo language description for conciseness.

3.1 Definition

A COMET *component*, as shown in Figure 1, is a bulk of code with a well defined interface. This interface can be described through *attributes*, *use/provide*, *MPI*, and *data* ports. *Attributes* describe values, fixed at runtime to configure the component. *Provide ports* describe services provided by the component (typically an object-oriented interface with a set of methods). *Use ports* describe services required by the component and provided by others. *MPI ports* support sharing of a MPI communicator between components.

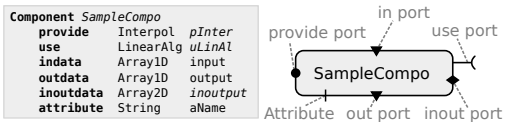


Figure 1: Example of a component definition (left) and a graphical representation (right).

Figure 2: Simplified example of metatask definition (left), a graphical representation (middle) and a possible produced task graph by the metatask (right).

In addition to L^{2}C, Comet components may also expose data ports. *Data port* express interactions through data buffers of a well defined type (*e.g.,* 2D array). Such an interaction has an access mode that can be: `in` for read-only access, `out` for write-only access or `inout` for read-write access.

Data based interactions enable components to send/receive data to/from *metatasks*. Components can poll the data port state (either available or busy) or wait for the data to be available. A data port is available when its buffer is not used by any other entity (component or metatask) for write mode data port, and when data is ready for read mode data port.

A *metatask* is a unit of composition describing bags of (independent) tasks executing the same code. Like a component, such a unit has a well defined interface. It may include *parameters* and *data ports*. A parameter is a typed value, fixed at compile-time, used to configure the metatask. A metatask also has an implicit use port named `compute` that requires a service providing the task implementation.

Figure 2 displays a simple example of metatask definition with only one inout data port. The figure also shows the task graph that is implied by such a metatask: rectangles are data and circles are tasks. This is a very simple example of metatask that produces only one task that reads and writes in the same data buffer.

To generate a bag of tasks, a metatask requires at least one *partitioned* data as input that adds partitioning information to a data buffer. The buffer is split into disjoint data *fragments* with typically one task per fragment so that the partitioning controls the number of tasks. Comet does not define a list of valid partitioning. It only requires all fragments of a buffer to be indexed by a local fragment id (for example an integer between 1 and the number of fragments). For example, examples in this paper use a block partitioning of multidimensional arrays.

When a metatask receives multiple partitioned data as input, one must determine which fragments from each buffer to combine for each task. This is solved in Comet by extending the metatask definition with a *relation expression* that describes how to match fragments of data ports. This can be very complex in the general case and we therefore design the *relation language* support to be extensible so as to use the best suited domain specific language (DSL).

This paper makes use of three relation languages associated to multidimensional arrays: `identity`, `frag-align`, and `frag-transpose`. The `identity` language requires no end-user expression and matches fragments with the same index as illustrated in Figure 3. It does however require all partitioned data of the metatask to contain the exact same number of fragments. The two other languages are presented in Section 4 as relation languages are not directly part of the core of the Comet model.

The description of a Comet application is done within a Comet *assembly*. Such an assembly contains component instances, dataflow sections, connections, and an entry point.

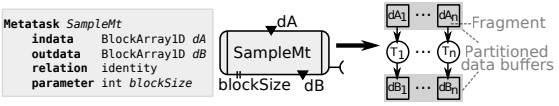


Figure 3: Example of metatask definition with two data ports (left), a graphical representation (middle), and a possible produced task graph (right).

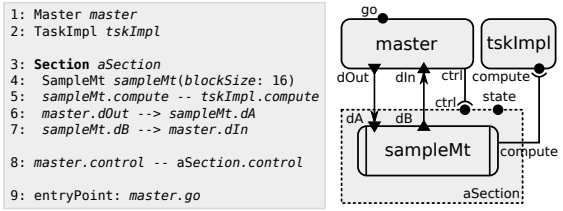


Figure 4: Example of a COMET assembly. Line 1-2: instantiation of two components. Line 3-7: declaration of a dataflow section. Line 4: instantiation of a metatask. Line 5 & 8: use-provide connections. Line 6-7: data connections. Line 9: Assembly entry point declaration.

A *dataflow section* contains a group of interconnected metatasks. The main purpose of this section is to control the submission of tasks produced by metatasks: a dataflow section exposes a service to start the dataflow and another to poll its state (either idle, submitting or running). Once components have correctly set input data ports of a dataflow section, it is the responsability of a component connected to such a port to actually launch the execution of a dataflow section.

Connections in an assembly include use-provide connections and data connections. Use-provide connections connect a use port to a provide port. The component instance exposing the use port can use the service provided by the provide port (it can invoke its methods.) Data connections are oriented. They connect data output ports (*i.e.,* out or inout) to input ports (*i.e.,* in or inout). This means that the content of data buffers flow from the source data port to the destination. The assembly entry point references a provide port used to start the execution of the assembly.

Figure 4 displays an example of an assembly with two components and a section containing a metatask. The execution of this assembly works as follow. First the master component configures its `dOut` data port with a valid data buffer. Then it starts the section through its `ctrl` use port and waits for the section to be completed (*i.e.,* for all tasks to be executed.) The data is partitioned in fragments. The metatask `sampleMt` schedules the tasks based on their fragment dependencies. Each execution of a task invokes the service `compute` provided by `taskImpl`. Once the dataflow section is completed, the master component exits from its waiting so that it can retrieve the computed data through its `dIn` data port.

Whenever two connected metatasks work on a same data buffer with different partitionings, the data buffer must be repartitioned. As COMET is defined for shared memory, repartitioning consists in defining the dependencies between tasks working on fragments of the two partitionings of a single data buffer. This operation is required to enforce a correct execution order. Multiple repartitioning strategies are possible: from a global barrier potentially causing over-synchronization to exact dependencies computation potentially resulting in a large amount of dependencies that can induce overheads. This partitioning strategy is therefore not specified in the COMET model and left as an implementation choice so as to make it possible to select the best choice in each situation. Section 5 evaluates two strategies (barrier and exact dependencies.)

Figure 5 presents an example of dataflow section involving repartitioning between two metatasks

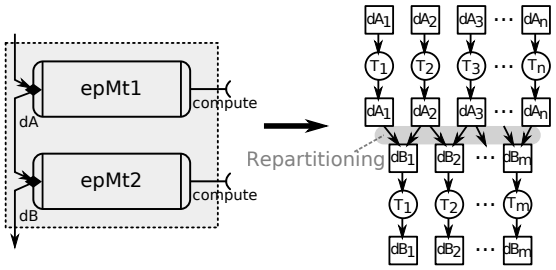


Figure 5: Example of a partial dataflow section of an assembly with a data repartitioning between two metatasks (left) and a possible produced task graph (right).

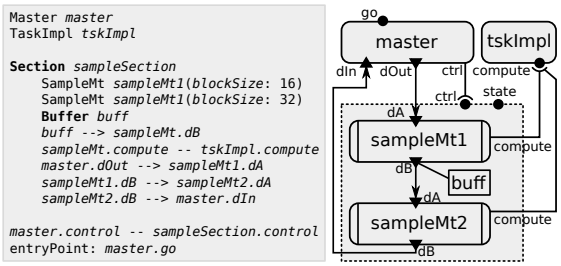


Figure 6: Example of a temporary buffer declaration (`Buffer` keyword) and usage. Cf Figure 4 for more information on the assembly elements.

performing embarrassingly parallel computations with different partitioning. The figure also displays the logical implied task graph. From the user point of view, the repartitioning is transparent and does not require any effort (neither in the assembly or component implementations). At runtime, the repartitioning strategy creates dependencies to enforce consistency between tasks of `epMt1` and `epMt2` that work on overlapping fragments.

The results of a metatask can be stored in temporary data buffers. This makes data buffer reuse possible by one or multiple tasks of the same section. Moreover, fragments can be created on demand according to the scheduling, since temporary buffers can be also partitioned. Such a buffer needs to be declared in the dataflow section. It can be connected to `in` and `out` data ports (not `inout`) using a data connection. The life-cycle of a temporary data buffer is automatically managed by its dataflow section.

Figure 6 gives an example of an assembly with a temporary buffer. In this example, `sampleMt1` tasks write in temporary fragments (named `buff`) that can be directly reused by tasks of `sampleMt2` and then automatically deleted or recycled by the runtime.

In order to be used in tasks, use and provide ports have to be manually tagged as *task-safe*. Since tasks can be executed concurrently, task-safe ports have to be reentrant to support being called from a task execution context. A task-safe use port can only be connected to task-safe provide port.

3.2 Discussion

From the point of view of a metatask, the semantics of declaring an *in* and an *out* data ports differs from declaring an *inout* data port. Indeed, an *inout* port implies an in-place computation while the combination of an *in* and *out* port implies out-of-place computation. Hence, the model forbids data buffer aliasing for correctness of computation.

COMET aims at supporting legacy code. However, there is no easy mechanism to guarantee that task-safe provide port implementations are actually task-safe. It is up to the developer to respect the contract of safety of component interfaces.

Metatasks define bag of independent tasks. These tasks usually depend on tasks submitted by other metatasks. Consequently, parallel patterns such as reduction or scan are not possible at dataflow level with the current COMET model. Reductions have to be done within a component. The required COMET extension of the dataflow section to support such patterns is left for future work.

4 Prototype Comet Implementation

The COMET execution platform is made of a dedicated source-to-source compiler and runtime. The design of these tools follow the philosophy outlined in [5]. This implementation is based on L^2C for components and OpenMP for dependent tasks [22].

4.1 Compiler Overview

The compiler takes as input a COMET assembly description file (an XML dialect) and C++ component implementations (L^2C component extended with data ports). It generates a L^2C assembly including both the application and runtime components. Some of the runtime components are generated by the compiler while other are predefined. In particular, the compiler generates one component per dataflow section. This component submits the tasks associated to its dataflow section at runtime thank to OpenMP directives.

The COMET compiler relies on some expert oriented configuration files for the definitions and implementations of plain data types, partitioned data types, and (re)partitionings. Implementations are provided though L^2C components and are made available in a separate repository enabling the COMET implementation to be easily extensible.

The current compiler supports multi-dimensional arrays that can be partitioned in fixed-size blocks and repartitioned either using barriers or fine-grained dependencies. It supports three relation languages: identity, frag-align, and frag-transpose. Additional languages can be supported; however it currently requires to modify the compiler. The addition of relation languages without modification of the compiler shall be the focus of future work.

As explained in Section 3.1, the identity language linearizes array fragments and then matches all i_{th} elements of the resulting ordered sets. The frag-align language assumes fragments are aligned: data buffers must have the same size and partitioning. An associated expression is composed of output fragments (left side) and input fragments (right side). Fragments are indexed by implicit variables, plus optionally an integer. All possible output fragments are iterated to generate tasks. Such an alignment language supports the description of stencils. An example of valid expressions is: $f_{out}(i) \leftarrow f_{in}(i-1), f_{in}(i), f_{in}(i+1)$. The frag-transpose language is similar to frag-align except that it does not handle relative indexing but allows alignment with transposed data. An example of valid expression is: $f_{out}(i,j) \leftarrow f_{in}(j,i)$.

4.2 Runtime overview

The runtime relies on some components for managing data, partitioned data, repartitionings, and relation languages. The runtime uses the architecture described in [5] and this section only provides a rough overview of its main components.

Data components manage data life-cycle. They are responsible for the memory allocation and provide a unique buffer reference to carry dependencies during partitioning operations.

Partitioned data manage partitioned data (*i.e.* fragments) life-cycle and handle dependencies between data buffers and their fragments (for partitioning and unpartitioning.) They support unique references for fragments enabling tasks to work on them and all kinds of partitioning operations.

Repartitioning component are responsible for carrying dependencies between fragments. This is actually achieved using empty OpenMP tasks with dependencies containing fragment references over both the source and destination partitioned data buffers. When no repartitioning component is available, the implementation may choose either an OpenMP task barrier, or an unpartitioning of the last partitioned data buffer followed by a partitioning of the new one (local synchronization).

5 Evaluation

The evaluation of a component framework such as Comet is tricky. Because of the absence of consensual benchmark suites in the research community about software component model and HPC, we made the evaluation over five synthetic benchmarks with computation patterns inspired from HPC applications rewritten with Comet.

The target architecture is common to OpenMP which mainly consists of shared memory architecture. The necessary extensions to OpenMP API to efficiently exploit NUMA architecture are available in various OpenMP research runtimes: Qthread [21], OMPss [11] or those [26] which are present in the KOMP OpenMP runtime selected to conduct our experimentations. Nevertheless production level OpenMP runtimes (GNU or Intel runtimes for instance) do not include them because of the absence of standardized API in OpenMP. Thus, we restrict ourself to use a subset of the architecture in order to avoid NUMA effect.

This section firstly presents the benchmarks. Then, it deals with separation of concerns as a software engineering property. Last, it evaluates the performance of Comet versions in particular by comparing them to hand written OpenMP versions of the benchmarks.

5.1 Benchmark presentation

Three base selected uses-cases of the benchmark are EP (for embarrassingly parallel), ST (for stencil), and TRANSP (for transpose). They represent common patterns in HPC applications for example found in NAS benchmarks (EP and FT).

EP is a memory-bound in-place 2D embarrassingly parallel code, also presents in NAS EP. The Comet assembly makes use of 2D block data partitioned size $b \times b$ with a task implementation that makes one floating point operation per point. ST is a 4-point 2D Jacobi stencil, also memory-bound. Data is distributed using 2D blocks of size $b \times b$. The task implementation reads 5 blocks and writes one data block in a distinct buffer. TRANSP computation consists in two line-wise compute-bound computations, interleaved by two memory-bound transpositions of the array. This pattern is frequently used by multidimensional fast-Fourier transforms (FFT) such as in NAS FT. The whole computation is performed in-place.

Moreover, we evaluate composition and data redistribution interests of Comet by adding two other use-cases in the set of benchmarks: EP-EP as a sequence of two memory-bound in-place 2D EP codes and ST-ST as a sequence of two ST codes.

Figure 7 shows the task graphs of the five benchmarks. Repartitioning is displayed when it occurs. All the benchmarks operate with a regular-size block partitioning.

For each use-case, we have manually written a *reference implementation* using directly OpenMP without any Comet generated code. It has the same tasks and dependencies as those performed

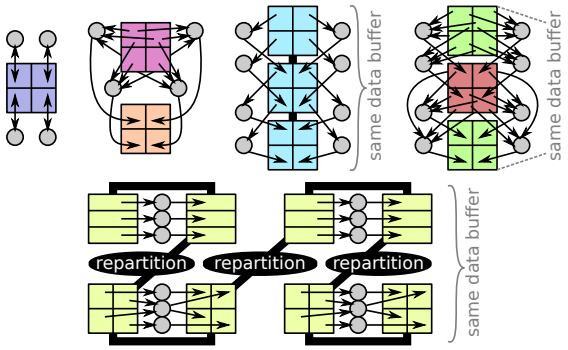


Figure 7: Task graph structure for the five benchmarks. From left to right and top to bottom: EP, ST, EP-EP, ST-ST, TRANSP.

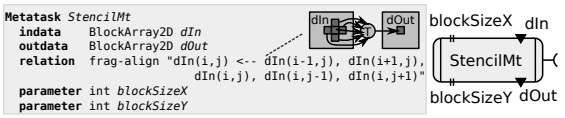


Figure 8: The metatask declaration of the ST use-case.

at runtime by the COMET assembly. The goal is to evaluate the runtime overhead introduced by COMET.

The COMET description of the EP use-case looks like the example provided in Figure 4 of Section 3, but it uses inout ports rather than in and out ports and it uses 2D buffers instead of 1D buffers.

The ST assembly is exactly the one presented in Figure 4 while the description of its metatask is given in Figure 8. Input and output data are partitioned in block of size $b \times b$. The relation expression in frag-align language generates as many tasks as there are output fragments. Each task reads the neighborhood of a `dIn` fragment at (i, j) and writes the `dOut` fragment at (i, j) with i and j two implicit variables ranging over the fragment iteration space.

EP-EP and ST-ST use-cases reuse the metatask of respectively the EP and ST use-cases and their assemblies are very similar to respectively those shown in Figures 5 and 6 presented in Section 3. The first and second metatasks of such assemblies are latter called Mt1 and Mt2.

Figure 9 displays the assembly of the TRANSP use-case while Figure 10 shows the description of the transposition metatask. This metatask transposes a whole data in-place. Data are partitioned in square blocks. The frag-transpose relation language enables the transposition of two fragments. For each task, a pair of fragments is both read and written: the ones at (i, j) and at (j, i) for all possible i and j such that $i \geq j$. It generates as many tasks as there are pairs of fragments. Diagonal fragments are duplicated from the user point-of-view.

5.2 Separation of concerns

From a software engineering point of view, COMET enables the independent definition of application building-blocks. Indeed, COMET units (components and metatasks) have well-defined interfaces. Interfaces are their only possible interaction points. Thus, dependencies between COMET units are only and explicitly described through port connections in an assembly. For example, in the TRANSP use-case, the transposition metatask does not depend on the other metatask: it contains only data ports and a relation expression, while the component that de-

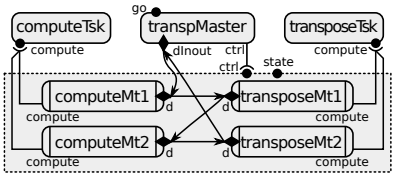


Figure 9: The COMET assembly of the TRANSP use-case.

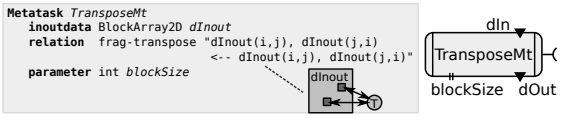


Figure 10: A transposition metatask of the TRANSP use-case.

fines its implementation is only dependent of the metatask interface. Thus adaptation of the transposition implementation requires no understanding of other units of the use-case.

COMET metatasks can be composed together, despite the use of different data partitioning. This can be seen on the TRANSP use-case. On one hand, compute-bound metatasks work on line blocks, since the computation needs to operate on the whole dimension. On the other hand, transposition metatasks work on square blocks, since the whole transposition must be performed in-place and should run in parallel to be efficient. Many HPC applications makes use of barriers or hand-tune the execution order of computational parts. For example, a barrier can be inserted between compute-bound codes and transpositions. Another possibility is to mix compute-bound with transpositions so that their execution is interleaved. The former may be detrimental to performances by over-synchronizing cores. The latter is detrimental to the separation of concerns by mixing independent codes, then it is more difficult to maintain. Neglecting separation of concerns often raises application maintainability costs (*e.g.,* more difficult to support new computational methods or new platforms).

COMET aims at solving this issue by using transparent repartitioning: when multiple data ports make use of the same data type but with different partitionings, the implementation can either makes use of barriers or fine-grain task dependencies. As a result, codes can still be independent and their efficient composition is delegated to experts (implemented in third-party components). The same problems occur in the EP-EP and ST-ST use-cases, since computations may have different computational intensity.

5.3 Performance Evaluation

This section deals with the performance evaluation of COMET on the five use-cases. Performance is evaluated on a socket of a shared memory node of the LIP laboratory (Lyon, France). A socket contains 24 cores Broadwell Intel Xeon E7-8890 v4 (2.20GHz). Each experiment has been done 20 times and the median is displayed. Error bars on plots are not visible since errors are always negligible (less than 1%). The compiler used is GNU GCC 6.3 with -O3 optimization option. Generated COMET assemblies target standard OpenMP C++ compiler and runtime. Nevertheless, we mainly experiment with the new implementation of KOMP [10] based on a fork of the Intel runtime implementation (cf `http://gitlab.inria.fr/openmp/libkomp`). It includes extensions to fully support dependent GOMP task, to add tracing and performance tool. It provides task management using the original Cilk work stealing T.H.E. implementation [14].

Some comparisons between the GOMP GNU OpenMP runtime and KOMP are also presented

Y		EP			SP		
Size	#tasks	Ref.	Comet	Ratio	Ref.	Comet	Ratio
1024	2^6	0.399	0.399	0.999	0.601	0.603	1.003
512	2^7	0.401	0.401	0.999	0.607	0.607	0.999
256	2^8	0.399	0.399	1.000	0.604	0.603	1.000
128	2^9	0.399	0.399	1.000	0.603	0.605	1.003
64	2^{10}	0.399	0.400	1.001	0.602	0.604	1.003
32	2^{11}	0.401	0.401	1.001	0.604	0.604	1.001
16	2^{12}	0.406	0.403	0.992	0.609	0.612	1.005
8	2^{13}	0.405	0.407	1.006	0.705	0.702	0.996
4	2^{14}	0.590	0.538	0.912	1.156	1.206	1.044
2	2^{15}	1.152	1.071	0.930	2.226	2.336	1.049
1	2^{16}	2.321	2.121	0.914	4.479	4.647	1.038

Table 1: Completion time (in second) and ratio for the reference and COMET versions of the EP and ST use-cases over varying block sizes (1024xY) for a 8192x8192 data.

to motivate the interest of small overhead in fine grain tasks to accelerate computation.

5.3.1 Task scalability

Table 1 reports the completion time of the EP and ST use-cases over different block sizes with a data size of 8192x8192.

First, the completion time grows when the bloc size decreases. This due to the task overhead caused by the KOMP OpenMP runtime that becomes significant for a large number of tasks: 65536 tasks are generated for the block size 1024x1. Original Intel or GNU GOMP have the same behavior, with a larger task overhead for GOMP. Moreover, a detailed analysis of the execution trace shows that this is due to the sequential task submission.

Nevertheless the COMET implementation obtains similar performance as the reference OpenMP implementation. COMET is up to 9 % faster than the reference version on the EP use-case and 5 % slower on the ST use-case. The bigger difference is when the number of tasks is important and the underlying OpenMP task management overhead becomes predominant. The overheads of COMET over the reference OpenMP implementation are negligible (less than 1%) for 4096 tasks and below (block size bigger than 1024x16 for this instance).

		EP		ST	
Size	#tasks	μs	ratio	μs	ratio
4096 x 4096	128	843	1.0008	1,210	1.0006
8192 x 4096	256	835	1.0004	1,130	1.0016
8192 x 16384	1024	826	1.0043	1,180	1.0003
16384 x 16384	2048	827	1.0001	1,140	1.0032

Table 2: Time per task and ratio of COMET time over reference implementation time for EP and ST for several configurations of data size with a fixed-size block of 1024x128.

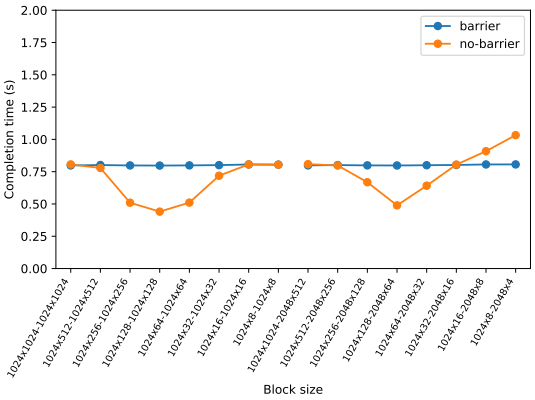


Figure 11: Completion time of the EP-EP use-case over different block sizes with a data size of 8192x8192. Block sizes of the metatasks Mt1 and Mt2 are of the form $B_{Mt1} - B_{Mt2}$. Left: without repartitioning. Right: with repartitioning.

5.3.2 Data-size scalability

Table 2 reports the time per task and the ratio between the completion time of COMET over the reference version on the EP and ST use-cases for different data sizes (from 4096x4096 up to 16384x16384) with a fixed-size block of 1024x128.

The COMET implementation is as fast as the corresponding reference implementation with a ratio very close to 1: COMET does not add any overhead compared to a hand-written OpenMP implementation. Moreover, the time per task with COMET remains stable while increasing the problem size and keeping work per task constant (fixed block 1024x128), *i.e.,* a form of weak scaling.

5.3.3 Harnessing cache-reuse

Figure 11 shows the completion time of the EP-EP use-case using COMET over different block sizes, with a data size of 8192x8192, with and without a barrier (cf Section 4.2). The left-hand side contains cases without a need for repartitioning between the two EP codes (same block size) and the right-hand side cases require repartitioning due to different block sizes between the two EPs. The block sizes of the first EP is denoted B_{Mt1} and B_{Mt2} for the second EP.

The version with no barrier always outperforms the one with a barrier if there is no need for repartitioning (Fig. 11, left). It is at most 81% faster. Faster completion times comes from cache data reuse between tasks generated from the metatask Mt1 and those of Mt2 when fragments are sufficiently small to fit into cache. In fact, due to the absence of barrier, the task scheduler is able to follow the dependencies between tasks from Mt1 and Mt2: once last predecessor Mt1 task ends, it makes ready Mt2 tasks. A contrario, the barrier version forces the computation of all the tasks of Mt1 before Mt2 and, because the whole data can not fit in the cache, no cache reuse is possible.

Faster times are observed when there are enough tasks to avoid starvation due to work imbalance and when the number of tasks is not too huge. In fact, as discussed above, the task submission is slow (sequential) and it may act as a barrier if Mt1 tasks are completed before Mt2 tasks are submitted: the scheduler is not able to take advantage of task dependencies.

With repartitioning (Fig. 11, right), the same effects are observed: the no barrier version is

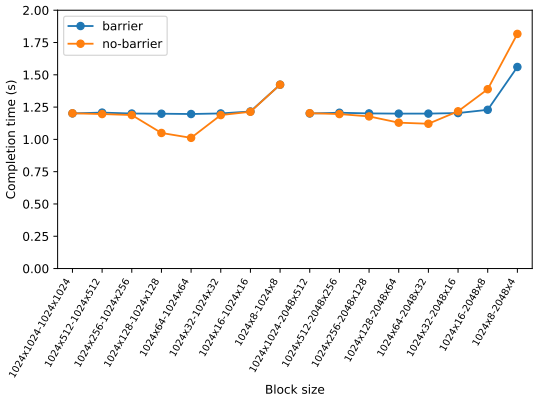


Figure 12: Completion time of the ST-ST use-case over different block sizes with a data size of 8192x8192. Block sizes of the metatasks Mt1 and Mt2 are of the form $B_{Mt1} - B_{Mt2}$. Left: without repartitioning. Right: with repartitioning.

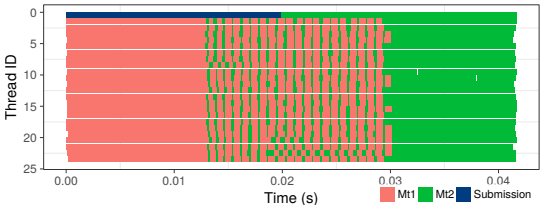


Figure 13: Task schedule of one iteration of the ST-ST use-case obtained with a block size 1024x64 and a data buffer size of 8192x8192.

at most 63% faster, except for fine-grain block size where it is up to 28% slower. At this extrema when block size is too small, the completion time is bounded by the task submission time that is slowed down by additional repartitioning tasks.

Figure 12 reports the results obtained with the ST-ST use-case in the similar way than the EP-EP experiments presented in Figure 11. Overall, the version with no barrier is up to 18% faster without repartitioning and 7% faster with repartitioning. This is compliant with EP-EP results. However, the gain of suppressing the barrier is less important because the dependencies between Mt1 tasks and Mt2 tasks are more complex in ST, letting few possibility to schedule tasks following dependencies to increase in-cache data reuse.

Figure 13 shows the task schedule of one iteration of the ST-ST use-case obtained with a block size 1024x64 and a data buffer size of 8192x8192. At the middle of the schedule, in-depth scheduling is performed and Mt2 tasks are faster than at the end since they reuse in-cache data. This does not happend before because the submission phase takes some times and thus it prevents task to be scheduled in-depth earlier.

5.3.4 Enabling memory access by computation overlap

Figure 14 shows the completion time of the COMET TRANSP use-case with barrier and no barrier versions for different block sizes. The data buffer size is 8192x8192.

First, The version with no barrier is alway faster than the one without: from up 9% up to 17%. This is due to two factors: work imbalance and memory throughput utilization. Indeed, with few

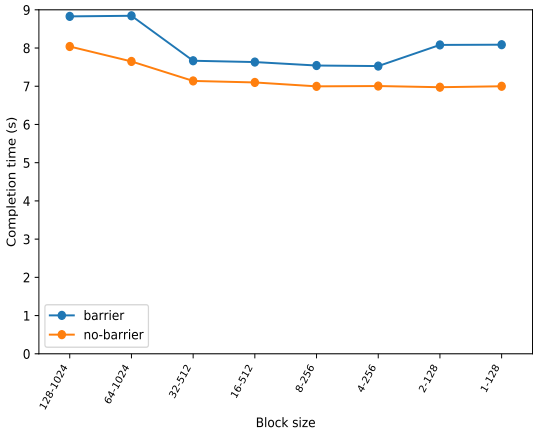


Figure 14: Completion time of the TRANSP use-case over different block sizes with a data buffer size of 8192x8192. Block sizes are under the form $B_{compute} - B_{transpose}$ where $B_{compute}$ is the line block height of MtCompute and $B_{transpose}$ is the block size for MtTranspose.

tasks workers are starving due to a work imbalance. In the no barrier version, the composition generated by COMET exploits the real dependencies between the tasks from different metatasks. The scheduler is able to overlap the execution of all tasks, which reduces starvation.

With a lot of tasks, memory-bound transposition tasks are performing faster without a barrier since they can be overlapped with compute-bound tasks. This effect is visible on the completion time breakdown shown in Figure 15 for the Tr1 and Tr2 tasks. For all the configurations where the barrier version is used, the times for Tr1 or Tr2 are inflated in comparison of the no barrier version. If a barrier is used, the scheduler runs, on all the cores, all the tasks Tr1 or Tr2 which imposes a high pressure in the memory traffic. Due to contention in the traffic, memory accesses are longer and thus completion time increases. In the no barrier versions, the scheduler overlaps computation tasks (Mt1 and Mt2) with tasks making only memory accesses (Tr1 and Tr2) which decrease the pressure on the memory subsystem as memory accesses are spread over a longer time.

5.3.5 Performance over the GOMP runtime

Results obtained with the GOMP runtime are similar to those obtained with the KOMP runtime on task scalability experiments. However, GOMP introduces a higher task overhead causing the completion time to skyrocket up to 14.1 s and 14.8 s respectively on the EP and ST use-cases instead of 2.1 s and 4.6 s.

Composition experiments behave differently with GOMP than with KOMP. Figure 16 and 17 on next page respectively present the completion time of the EP-EP and ST-ST use-cases using GOMP with the same conditions that Figures 11 and 12. Overall, the version with no barrier is slightly faster than the one without on the EP-EP use-case: up to 13% without the need for repartitioning and from 27% slower up to 11% faster with repartitioning. On the ST-ST use-case, the performance of the two versions are very close. Without considering outliers, the version without a barrier is 3% faster than the one with no barrier, both with and without repartitioning. With GOMP there is a small gain to avoid barrier during composition. Thus is due to limited cache reuse as previously discussed.

TRANSP results with GOMP are similar to those with KOMP where the version without

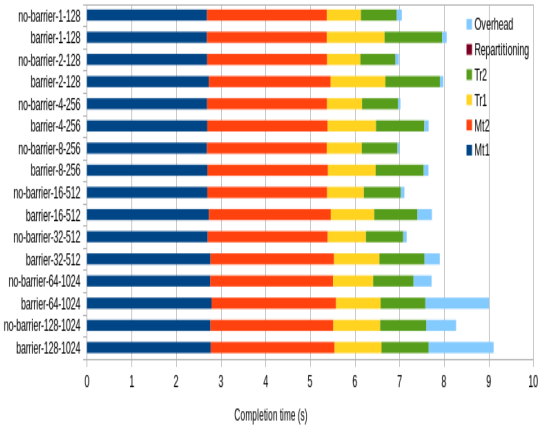


Figure 15: Completion time breakdown of the TRANSP use-case over different block sizes with a data buffer size of 8192x8192.

barrier is always faster than the one without: from 4%, up to 16%. GOMP well dispatches memory accesses by overlapping them with computation.

5.4 Discussion

From previous results, we can conclude that the performance of COMET is equivalent to hand-written OpenMP reference implementation of the use-cases when simple computation patterns are used. With increasing complexity of the patterns, especially in presence of the compositions, COMET demonstrates its superiority: the expression of the composition remains simple while the framework can automatically generates efficient execution using point-to-point synchronisations between tasks without any global synchronization (OpenMP barrier). This solution is very effective, even for small memory bound tasks, to harness cache reuse and to overlap tasks across redistribution boundary with high level compositions of codes.

Results with the two production level OpenMP runtimes were, at a first glance, disappointing. GOMP, or Intel runtime, whose results are not presented here, are unable to benefit from the barrier eviction. Several assumptions were formulated before we focus on the cutoff strategy to reduce task overhead in these runtimes. By serializing execution of task at creation, it make coarsen granularity of the tasks [12]. The impact on the task scheduling is negative and acts like to the use of a barrier. Indeed tasks that may harness cache reuse are *de facto* submitted when most previous tasks have been executed. In KOMP, the Cilk T.H.E work stealing scheduling is based on unbounded queues of tasks that better fit the COMET component composition assumption.

It becomes challenging to implement COMET composition on top OpenMP runtime in portable way. Besides task scheduling problem, OpenMP runtime badly sustains important number of (fine grain) dependent tasks. GOMP task overhead is bigger than the Intel OpenMP runtime extended by KOMP and it limits the scalability of our proposition. Dependencies are the source of overhead. Their definitions, thus their computations, can not be parallelized with OpenMP due to limitation in the specification [22] that imposes a sequential creation of dependent tasks. In order to overcome this limitation, we are looking into exporting runtime functions to bypass computations of dependencies if the relation language (cf Section 3.1) permits to compute them at compilation step of the COMET assembly description.

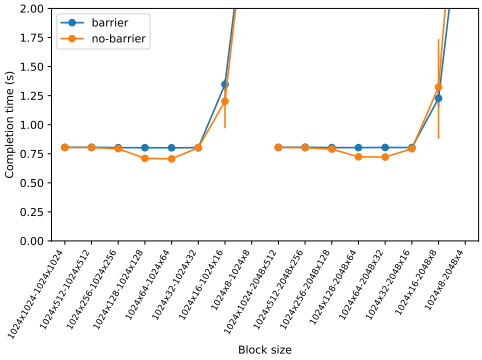


Figure 16: Completion time of the EP-EP use-case for different block sizes with a data size of 8192x8192 using the GOMP runtime. Block sizes of the metatasks Mt1 and Mt2 are of the form $B_{Mt1} - B_{Mt2}$.

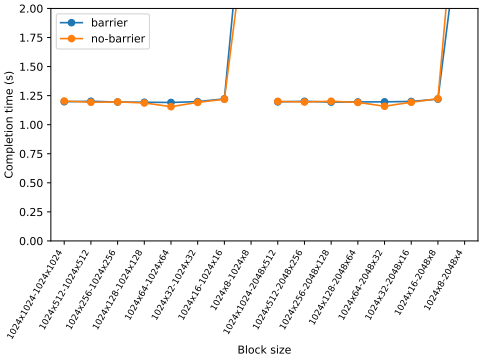


Figure 17: Completion time of the ST-ST use-case for different block sizes with a data size of 8192x8192 using the GOMP runtime. Block sizes of the metatasks Mt1 and Mt2 are of the form $B_{Mt1} - B_{Mt2}$.

Let us finish this section by remarking that we only make experimentations using the maximal number of cores to avoid NUMA effect (see the introduction of this section). This is not the optimal number of core for memory bound applications where memory contention occurs, as for instance in results presented on TRANSP. This number has been fixed for three main reasons: i) Several real OpenMP applications make use of all the reserved cores; ii) With less cores, COMET framework will be able to exploit cache reuse because the OpenMP tasks' scheduler will have access to the whole task dependencies more quickly, relatively to our experimental conditions; iii) Making experiments with the maximal number of cores while keeping the capacity to reschedule fine grain tasks across composition boundary, is a necessary step to further exploit many-core architectures.

6 Conclusion and Future Work

Programming parallel applications is difficult. It is even more difficult if software engineering issues such as separation of concerns have to be taken into consideration to improve maintainability, portability, and reduce the cost of development (increasing code reuse, simplifying code substitution, etc,). Programming models shall relieve such a burden from developers. The challenging issue in HPC is to let applications still be able to achieve high performance.

A promising approach is brought by software component models that are built on separation of concern concepts (code reuse). Nevertheless, existing component models were specialized either to structural composition (CCA, L^{2}C) or to dataflow composition.

This paper has presented the COMET programming model that merges these two forms of composition within a coherent and efficient model. Advanced code compositions, based on partitioned data, enable to easily describe some common HPC patterns while enabling efficient execution thanks to an interleaving of different tasks. The efficiency of the model has been demonstrated on five synthetic benchmarks (EP, ST, TRANSP, EP-EP, ST-ST).

However, more work is needed. NUMA multi/many-core architecture, like the Intel KNL, must be targeted: the COMET runtime may manage effectively memory affinity thanks to high level information such as data distribution. Collective operations such as reductions have to be

inserted into the model. The question of adding control structures such as loops or conditionnals to dataflow sections is open. Relation languages also need more research: more parallel patterns as well as applications have to be analyzed to understand to which level it is possible to keep relation languages that enable efficient execution. Another direction of work is enabling automatic launches of dataflow sections as well as the support of stream based computations.

Acknowledgment

This work was supported by the PIA ELCI project of the French FSN and by the Energy oriented Center of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*. Wiley, October 2014.

[2] B. A. Allan and al. A Component Architecture for High-Performance Scientific Computing. *International Journal of High Performance Computing Applications*, 2006.

[3] Jeremie Allard, Valerie Gouranton, Loick Lecointre, Sebastien Limet, Emmanuel Melin, Bruno Raffin, and Sophie Robert. FlowVR: a middleware for large scale virtual reality applications. In *International Euro-Par Conference on Parallel Processing*. Springer, 2004.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 2011.

[5] Olivier Aumage, Julien Bigot, Hélène Coullon, Christian Pérez, and Jérôme Richard. Combining Both a Component Model and a Task-based Model for HPC Applications: a Feasibility Study on GYSELA. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).*, Madrid, Spain, May 2017.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Intl Conf. on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[7] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Andrew Richards, Raymond Namyst, Beverly Bachmayer, Christoph Kessler, David Moloner, and Peter Sanders. The PEPPHER approach to programmability and performance portability for heterogeneous many-core architectures. In *ParCo*. IOS press, 2011.

[8] Julien Bigot, Zhengxiong Hou, Christian Pérez, and Vincent Pichon. A low level component model easing performance portability of HPC applications. *Computing*, 2013.

[9] Hinde Lilia Bouziane, Christian Pérez, and Thierry Priol. A software component model with spatial and temporal compositions for grid infrastructures. In *International Euro-Par Conference on Parallel Processing*. Springer, 2008.

[10] François Broquedis, Thierry Gautier, and Vincent Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, IWOMP 2016*, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Javier Bueno, J. Planas, Alejandro Duran, Xavier Martorell, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. Productive programming of gpu clusters with ompss. In *International Parallel and Distributed Processing Symposium*. IEEE, 2012.

[12] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.

[13] Allan Espinosa, Pete Beckman, Mihael Hategan, Zhao Zhang, Michael Wilde, Kamil Iskra, Ian Foster, Ben Clifford, and Ioan Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 2009.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[15] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *International Symposium on Parallel and Distributed Processing*. IEEE, 2013.

[16] Virginie Grandgirard, Jérémie Abiteboul, Julien Bigot, Thomas Cartier-Michaud, Nicolas Crouseilles, Guilhem Dif-Pradalier, Ch. Ehrlacher, D. Esteve, Xavier Garbet, Philippe Ghendrih, Guillaume Latu, Michel Mehrenberger, C. Norscini, Chantal Passeron, Fabien Rozar, Yanick Sarazin, Eric Sonnendrücker, A. Strugarek, and David Zarzoso. A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 2016.

[17] Guillaume Latu, Julien Bigot, Nicolas Bouzat, Judit Gimenez, and Virginie Grandgirard. Benefits of smt and of parallel transpose algorithm for the large-scale gysela application. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '16, pages 10:1–10:10, New York, NY, USA, 2016. ACM.

[18] Kung-Kiu Lau, Lily Safie, Petr Stepan, and Cuong Tran. A component model that is both control-driven and data-driven. In *International Symposium on Component Based Software Engineering*. ACM, 2011.

[19] M. D. McIlroy. Mass-produced Software Components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[20] Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay Fornarino. A data-driven workflow language for grids based on array programming principles. In *Workshop on Workflows in Support of Large-Scale Science*. ACM, 2009.

[21] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[22] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 4.5, November 2015.

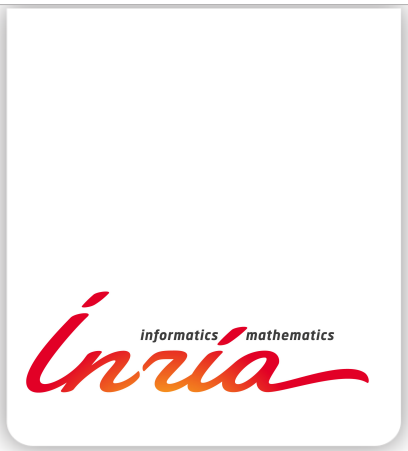
[23] Christian Pérez, Thierry Priol, and André Ribes. A parallel corba component model for numerical code coupling. *The International Journal of High Performance Computing Applications*, 17(4):417–429, 2003.

[24] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for hpc with logical regions. In *Intl Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015.

[25] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[26] Philippe Virouleau, Adrien Roussel, François Broquedis, Thierry Gautier, Fabrice Rastello, and Jean-Marc Gratien. Description, implementation and evaluation of an affinity clause for task directives. In Naoya Maruyama, Bronis R. de Supinski, and Mohamed Wahib, editors, *Proceedings of the 12th International Conference on OpenMP: Memory, Devices, and Tasks, IWOMP 2012*, pages 61–73, Cham, 2016. Springer International Publishing.

[27] Wei Wu, Aurélien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *International Parallel and Distributed Processing Symposium*. IEEE, 2015.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399